Table of Contents

Introduction	1.1
概述	1.2
SQLAlchemy ORM	1.3
对象关系入门-Object Relational Tutorial	1.3.1
配置映射-Mapper Configuration	1.3.2
配置关系-Relationship Configuration	1.3.3
载入对象-Loading Objects	1.3.4
使用对话-Using the Session	1.3.5
事件和内部实现-Events and Internals	1.3.6
ORM扩展-ORM Extensions	1.3.7
SQLAlchemy Core	1.4
SQL表达语言入门-SQL Expression Language Tutorial	1.4.1
SQL语句和表达式接口-SQL Statements and Expressions API	1.4.2
列和数据类型-Column and Data Types	1.4.3
引擎和连接使用-Engine and Connection Use	1.4.4
核心API基础-Core API Basics	1.4.5

sqlalchemy-docs-CN

1. 本项目为sqlalchemy1.1.1版本文档的中文译本，只是想通过翻译来熟悉一下sqlalchemy以及体验一下gitbook
2. 水平有限，不定期更新（可能每天更新一部分吧，flag已经立下了）
3. 如有问题，可以直接提issue
4. 遵循Apache License（开源协议选择建议可以看这里）
5. 项目已同步到gitbook

tutorial

发现翻译完好好久好久，开始使用的话实际上也不用这么麻烦，所以大家可以先看看我写的一个入门教程，基本上就可以简单的使用了。

Object Relational Tutorial(关系映射入门)

SQLAlchemy对象关系映射代表了用户使用Python定义类来与数据库中的表相关联的一种方式，类的实例则对应数据表中的一行数据。SQLAlchemy包括了一套将对象中的变化同步到数据库表中的系统，这套系统被称之为工作单元（unit of work），同时也提供了使用类查询来实现数据库查询以及查询表之间关系的功能。

SQLAlchemy ORM与SQLAlchemy表达式语言（SQLAlchemy Expression Language）是不同的，前者是在后者的基础上构建的，也就是说，是基于后者实现的（将后者封装，类似于实现一套API）。SQL表达式语言（SQL Expression Language）在 SQL Expression Language Tutorial一节有介绍，他代表了关系数据库最原始的一种架构，而SQLAlchemy ORM则代表了一种更高级也更抽象的实现，同时也是SQL表达式语言的应用。

尽管看起来ORM和表达式语言使用起来有点相似，但是其相似点比他们乍一看起来还要少（While there is overlap among the usage patterns of the ORM and the Expression Language, the similarities are more superficial than they may at first appear. 不是很确定翻译的是否正确，有问题可直接联系）。一种方式是通过透视用户定义的领域模型来实现数据的内容与结构，另外一种方式通过文字模式（ literal schema）和SQL表达式来明确的操作数据库（The other approaches it from the perspective of literal schema and SQL expression representations which are explicitly composed into messages consumed individually by the database.）

一个成功的应用可能只使用对象关系映射的构造。在更复杂的情况下，可能ORM以及SQL表达式语言互相配合使用也是必不可少的。

Version Check(版本检查)

使用以下代码检查SQLAlchemy版本

```
In [2]: import sqlalchemy
In [3]: sqlalchemy.__version__
Out[3]: '1.0.13'
```

PS: 安装SQLAlchemy过程，建议先安装virtualenv，再使用pip安装

```
pip install sqlalchemy==1.0.13  # 版本号可自选，文档对应的为1.1.0
```

---

connecting(连接数据库)

这篇教程中我们使用sqlite数据库来演示操作，我们使用 `create_engine()` 来连接需要操作的数据库：

```
In [1]: from sqlalchemy import create_engine

In [2]: engine = create_engine('sqlite:///:test:', echo=True)
```

`echo` 参数是用来设置SQLAlchemy日志的，通过Python标准库logging模块实现。设置为True的时候我们可以看见所以的操作记录。如果你在按照本教程进行学习，那么你可以将它设置为False来减少日志的输出。

`create_engine()` 的返回值是 `Engine` 的一个实例，此实例代表了操作数据库的核心接口，通过方言来处理数据库和数据库的API。在本例中，SQLite方言将被翻译成Python内置的sqlite3模块（个人理解，方言指的是每一种数据库的使用的方言，比方说mysql会有一种，sqlite又会有一种，而每种语言又会有很多在数据库的处理模块，比方说刚刚提到的Python内置的sqlite3模块）。

当第一次调用 `Engine.execute()` 或者 `Engine.connect()` 这种方法的时候，引擎（Engine）会和数据库建立一个真正的DBAPI连接，用来执行SQL语句。但是在创建了连接之后，我们很少直接使用Engine来操作数据库，更多的会使用ORM这种方式来操作数据库，这种方式在下面我们会看见具体的例子。

PS1: 什么是Database Urls

PS2: Lazy Connecting: 初次调用 `create_engine()` 的时候并不会真正的去连接数据库，只有在真正执行一条命令的时候才会去尝试建立连接。 目的是省资源，很多地方都会使用这种方式，比方说Python中的 lazy property

---

Declare a Mapping

当使用ORM的时候，配置过程以描述数据库的表来开始，然后我们定义与之匹配的类。在现在的SQLAlchemy中，这两个过程一般结合在一起，通过一个称之为声明（Declarative）的系统实现。这个系统帮我们定义类以及实现与表的对应。

声明系统实现类与表的对应是通过一系列基类实现的--即声明基类（declarative base class)。我们的应用程序经常只有一个此基类的示例。使用 `declarative_base()` 函数，如下：

```
In [2]: from sqlalchemy.ext.declarative import declarative_base

In [3]: Base = declarative_base()
```

有了"base"之后，我们可通过他定义任何数量的映射类。我们以一个user表来开始这个教程，user表记录了用我们应用的终端用户的信息。与之对应的类称之为User。表的信息包括表名，姓名，还有列的数据类型，如下：

```
from sqlalchemy import Column, Integer, String

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)

    def __repr__(self):
       return "<User(name='%s', fullname='%s', password='%s')>" % (self.name, self.ful
lname, self.password)
```

注： User class定义了一个 **repr()**函数，但是这并非必须的。使用它是为了在这个教程里使我们的例子（User对象）有更好看的格式。

使用Declarative定义的类最少要包括一个 **tablename** 参数，还有一列（作为数据库primary key）。SQLAlchemy自己从不做关于类所对应的表的任何假定，包括它没有任何于名称、数据类型或者约束内置的约束。但是并不代表没有一个规范的模式，相反，它鼓励你使用帮助函数和mixin类创建自己的自动规范约束，帮助函数和mixin类在Mixin和Custom Base Classes里有详细的描述。

当构建类的时候，Declarative使用特殊的Python accessor（即descriptors）代替了所有的Column对象；这个过程称之为instrumentation（中文意思：乐器）。这个"instrumented"过的类为我们的表提供了一种使用sql语句访问数据库中表的方式，同时也提供读取表中数据的方式（好特码复杂，原文：provide us with the means to refer to our table in a SQL context as well as to persist and load the values of columns from the database.）

除此之外，还有一些普通的Python类，可以让我们定义一些用于我们应用的一般的参数。

Create a Schema

通过Declarative系统构建了类之后，我们也定义了被称之为表的元数据信息。被SQLAlchemy用来为某特定的表呈现这些信息的对象被称之为Table对象，Declarative系统为我们实现了这些。我们可以通过检测 **table** 参数来查看这个对象：如下：

```
# 代码接上面
>>User.__table__
Table('users', MetaData(bind=None),
            Column('id', Integer(), table=<users>, primary_key=True, nullable=False),
            Column('name', String(), table=<users>),
            Column('fullname', String(), table=<users>),
            Column('password', String(), table=<users>), schema=None)
```

PS： 经典映射-Classical Mappings 虽然在Declarative系统里十分推荐使用，但是对使用SQLAlchemy ORM来说也并非必须的。在Declarative系统外，任何一个Python类都可以通过 `mapper()` 函数来映射到一个数据表，但是用的相对较少，可以在Classical Mappings看到介绍。

当声明一个类的时候，Declarative系统使用Python元类来实现，这样就可以在类被声明之后还可以执行一些额外的操作（不懂的同学可以看一下元类的左右）；在这个过程中，会根据我们的指定创建一个Table对象，然后通过构建一个Mapper对象使之与类关联。这个对象是一个幕后的对象（behind-the-scenes object），以至于我们一般不直接与他打交道（虽然在我们需要的时候它也能我们提供很多信息）

`Table` 对象是一系列的元数据的组合。当使用Declarative的时候，这个对象可以使用我们declarative的基类的 `.metadata` 属性（When using Declarative, this object is available using the .metadata attribute of our declarative base class.）。

元数据是一堆包含的可以在数据库里执行的命令集。因为我们的SQLite数据库目前还没有一个users表，我们可以使用这些元数据来创建这些表。下面，我们调用 `MetaData.create_all()` 方法来将这些传给数据库。然后就可以看到提交这些命令之后的过程，如下：

```
>>> Base.metadata.create_all(engine)
SELECT ...
PRAGMA table_info("users")
()
CREATE TABLE users (
    id INTEGER NOT NULL, name VARCHAR,
    fullname VARCHAR,
    password VARCHAR,
    PRIMARY KEY (id)
)
()
COMMIT
```

Minimal Table Descriptions vs. Full Descriptions

熟悉CREATE TABLE语法的同学可能注意到了我们刚刚创建VARCHAR列的时候没有指定长度，在SQLite和Postgresql里是允许的，但是在其他地方是被禁止的。所以如果你使用了其他的数据库同时想使用SQLAlchemy来创建一个表，那么可能需要给String type添加一个i长度，格式如下：

```
Column(String(50))
```

String和其他的Integer，Numeric等待都一样，除了创建表的时候其他时候用不到。

除此之外，Firebird和Oracle需要指定一个主key，但是SQLAlchemy并不会自动生存。对于这种情况，你可以使用 `Sequence` 来实现：

```
from sqlalchemy import Sequence
Column(Integer, Sequence('user_id_seq'), primary_key=True)
```

使用declarative来实现一个映射的完整应用如下：

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, Sequence('user_id_seq'), primary_key=True)
    name = Column(String(50))
    fullname = Column(String(50))
    password = Column(String(12))

    def __repr__(self):
        return "<User(name='%s', fullname='%s', password='%s')>" % (
                                self.name, self.fullname, self.password)
```

我们展示了这个完整的表以此对比来显示两者的区别（最小结构vs完整版），完整的可以用来完成创建某种特定的很严格的需求的表。

Create an Instance of the Mapped Class

声明完映射之后，我们来创建一个User对象，如下：

```
>>> ed_user = User(name='ed', fullname='Ed Jones', password='edspassword')
>>> ed_user.name
'ed'
>>> ed_user.password
'edspassword'
>>> str(ed_user.id)
'None'
```

注： **init**() 方法

我们使用Declarative系统创建的类包含了一个构造函数，可以自动的接受并匹配关键词。当然也可以自定义一个 **init**() 函数，这样的话就会覆盖自带的。

即使我们没有在构造函数里指定id，但是我们访问它的时候系统仍然给了一个默认的值（空值，和Python里其他的如果没有定义就会引起一个参数错误相反，这里不会有错误）。SQLAlchemy经常会在我们第一次使用的时候去给参数赋一个值。对于那些我们已经赋值的参数，仪器系统（instrumentation system）会在插入语句执行的时候跟踪这些参数（SQLAlchemy’s instrumentation normally produces this default value for column-mapped attributes when first accessed. For those attributes where we’ve actually assigned a value, the instrumentation system is tracking those assignments for use within an eventual INSERT statement to be emitted to the database.）。

Creating a Session

现在我们开始讨论数据库。ORM处理数据库的方式是通过Session来实现的。当我们第一次创建这个应用的时候，我们使用 `create_engine()` 语句，同时也定义一个Session类来当做一个工厂来处理Session对象（When we first set up the application, at the same level as our create_engine() statement, we define a Session class which will serve as a factory for new Session objects）：

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=engine)
```

如果此时你还没有一个engine对象，那么可以使用下面这种方法：

```
>>> Session = sessionmaker()
```

然后再创建一个engine（使用 `create_engine()` ）,然后使用configure来连接session和engine：

```
>>> Session.configure(bind=engine)  # once engine is available
```

这个定制后的类会创建一个绑定到我们指定数据库的Session对象，其他的交易特征也可以在调用sessionmaker的时候定义。这些东西会在下一章讨论。然后当你需要与数据库进行对话的时候，你需要创建一个Session实例：

```
>>> session = Session()
```

上面的Session是和我们的SQLite-enabled引擎相关联的，但是现在还没有与数据库链接。当他第一次被调用的时候才会建立与一个Engine维护的连接池连接，一直持续到我们关闭Session对象，或者提交完所有的变化。

PS: 会话的生命周期模式（Session Lifecycle Patterns）

说明时候创建一个会话依赖于我们在创建一个说明应用。记住，对话只是你对象指向一个数据库链接的一个工作空间，如果把对象进程比作一个晚宴，那么来宾的盘子还有盘子上的食物则是回话（数据库就是厨房？）！更多的了解请连链接—什么时候创建会话，什么时候提交，什么时候关闭。

添加／更新对象（Adding and Updating Objects）

为了持续操作(persist)我们的 `User` 对象，我们把他添加（ `add()` ）到会话中：

```
>>> ed_user = User(name='ed', fullname='Ed Jones', password='edspassword')
>>> session.add(ed_user)
```

现在，我们称这个对象是待定的（pending）；现在没有任何SQL语句被执行，同时这个对象也并代表数据库中的一行数据。对话（Session）会在需要的时候尽快持久化 `Ed Jones` ，这个过程称之为 **flush** 。如果我们在数据库里查询 `Ed Jones` ，所以的待定信息（pending information）都会首先被**flush**（冲刷？），随机查询请求被执行。

For example, below we create a new Query object which loads instances of User. We “filter by” the name attribute of ed, and indicate that we’d like only the first result in the full list of rows. A User instance is returned which is equivalent to that which we’ve added: 例如，下面我们创建一个 `User` 实例的查询对象，我们使用名字属性来过滤 `ed` ，然后只选取列表里的第一个数据。返回结果就是我们之前添加的那个 `User` 对象：

```
>>> our_user = session.query(User).filter_by(name='ed').first() # doctest:+NORMALIZE_W
HITESPACE
>>> our_user
<User(name='ed', fullname='Ed Jones', password='edspassword')>
```

实际上，会话（Session）已经识别出这行数据已经在内部的map对象中了（In fact, the Session has identified that the row returned is the same row as one already represented within its internal map of objects, so we actually got back the identical instance as that which we just added:），所以我们拿回的数据就是之前刚刚添加的那个：

```
>>> ed_user is our_user
True
```

ORM的概念在工作的地方会识别并且保证会是在一个特殊的行上。一旦一个对象已经在会话中有一个主键（primary key），所有关于这个key的SQL查询都只返回一个同样的Python对象，如果已经存在某个主键的对象，此时想添加一个同样主键的对象，就会引起一个错误。

我们可以使用 `add_all()` 函数一次性添加多个 `User` 对象：

```
>>> session.add_all([
...     User(name='wendy', fullname='Wendy Williams', password='foobar'),
...     User(name='mary', fullname='Mary Contrary', password='xxg527'),
...     User(name='fred', fullname='Fred Flinstone', password='blah')])
```

同样，我们如果觉得Ed的密码不太安全，也可以更改密码：

```
>>> ed_user.password = 'f8s7ccs'
```

而会话则时刻注意着这些变化，例如，他检测到了 `Ed Jones` 已经被更改了：

```
>>> session.dirty
IdentitySet([<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>])
```

同时，３个新的 `User` 对象处于等待提交的状态：

```
>>> session.new  # doctest: +SKIP
IdentitySet([<User(name='wendy', fullname='Wendy Williams', password='foobar')>,
<User(name='mary', fullname='Mary Contrary', password='xxg527')>,
<User(name='fred', fullname='Fred Flinstone', password='blah')>])
```

我们通知会话（Session）我们想保留所以的更改然后提交到数据库。使用 `commit()` 提交这些更改，会话提交 `UPDATE` 语句来更改 `ed` 的密码，同时使用 `INSERT` 语句来实现新的对象的插入：

```
>>> session.commit()
```

```
If we look at Ed’s id attribute, which earlier was None, it now has a value:
如果我们看Ed的id属性，之前我们没有设置，但是现在他有一个值了：
```

> > > ed_user.id # doctest: +NORMALIZE_WHITESPACE 1 ```

当会话在数据中中插入了新的行之后，所有新生成的标识符还有数据库为对象生成的一些默认值都是可以获取到的，同时是可以立即访问的或者可以一次获取到的（load-on-first-access）。在这种情况下，一行数据是重新加载的因为在提交之后就开始了一个新的事务。SQLAlchemy在新的事务中会从上一个事务中去获取更新的数据，这样大部分最近使用的语句都可获取到。重新加载的级别是可以定义的，可以去这里查看。

PS: 会话对象的状态－Session Object States

如果我们的 `User` 对象从 `Session` 外移到里边，为了真正的插入，他会经历三个状态（一共有四个状态）—短暂，等待，持久化（transient, pending, and persistent）。知道这几个状态的含义非常有帮助，推荐去这里详细理解一下。

回滚（Rolling Back）

既然会话在一个事务里边起作用，那么我们也可以在这个事务里回滚一些变化。让我们做两个更改，然后再删掉更改，把 `ed_user` 的用户名改成 `Edwardo`

```
>>> ed_user.name = 'Edwardo'
```

然后添加另外一个错误的假用户：

```
>>> fake_user = User(name='fakeuser', fullname='Invalid', password='12345')
>>> session.add(fake_user)
```

查询会话，我们可以看到这些已经放到了现在所处的事务当中：

```
>>> session.query(User).filter(User.name.in_(['Edwardo', 'fakeuser'])).all()
[<User(name='Edwardo', fullname='Ed Jones', password='f8s7ccs')>, <User(name='fakeuser
', fullname='Invalid', password='12345')>]
```

回滚一下，我们就看到 `ed_user` 的用户名已经改回了 `ed` ，假用户也不存在了：

```
>>> session.rollback()

SQL>>> ed_user.name
u'ed'
>>> fake_user in session
False
```

执行一个选择语句来看看对数据库的影响：

```
>>> session.query(User).filter(User.name.in_(['ed', 'fakeuser'])).all()
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

查询（Quering）

一个 `Query` 对象是通过 `Session` 里的 `query()` 方法来创建的。这个方法有一些参数，参数可以是一些内置的类或者描述符。下面我们声明一个 `User` 实例的查询对象。当我们执行这些语句的时候，就可以看到 `User` 对象返回来的查询列表：

```
>>> for instance in session.query(User).order_by(User.id):
...     print(instance.name, instance.fullname)
ed Ed Jones
wendy Wendy Williams
mary Mary Contrary
fred Fred Flinstone
```

```
```

> > > for name, fullname in session.query(User.name, User.fullname): ... print(name, fullname) ed Ed Jones wendy Wendy Williams mary Mary Contrary fred Fred Flinstone ```

`Query` 返回的结果称之为元组（tuples），通过 `KeyedTuple` class实现，同时可以被当做Python的原生对象来处理。参数的名称和参数一样，类名和类一样（不知道咋翻译，原文：The names are the same as the attribute’s name for an attribute, and the class name for a class，看例子理解的意思就是：row对应的是User, 想获取name就使用row.name，这样row和row.name分别都有其对应的User，User.name了）：

```
>>> for row in session.query(User, User.name).all():
...     print(row.User, row.name)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')> ed
<User(name='wendy', fullname='Wendy Williams', password='foobar')> wendy
<User(name='mary', fullname='Mary Contrary', password='xxg527')> mary
<User(name='fred', fullname='Fred Flinstone', password='blah')> fred
```

可以使用类元素衍生的一个对象 `lable()` 构造（construct）来给一列起另外的称呼，任何一个类的参数都可以这样使用（功能就像名字一样，打标签，起别名）：

```
>>> for row in session.query(User.name.label('name_label')).all():
...     print(row.name_label)
ed
wendy
mary
fred
```

这里把这个名字给了 `User` （实际上是给了User里的name），但是如果有两个参数呢（query()里有两个参数的情况，上面的例子只有一个）？可以使用 `aliased()` 来解决（和bash里的alias差不多）：

```
>>> from sqlalchemy.orm import aliased
>>> user_alias = aliased(User, name='user_alias')

>>> for row in session.query(user_alias, user_alias.name).all():
...     print(row.user_alias)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
<User(name='wendy', fullname='Wendy Williams', password='foobar')>
<User(name='mary', fullname='Mary Contrary', password='xxg527')>
<User(name='fred', fullname='Fred Flinstone', password='blah')>
```

```

```

> > > for u in session.query(User).order_by(User.id)[1:3]: ... print(u)
> > >
> > > ```

过滤结果使用 `filter_by()` 来实现，使用的参数是关键字：

```
>>> for name, in session.query(User.name).filter_by(fullname='Ed Jones'):
...     print(name)
ed
```

或者使用 `filter()` ， `filter()` 使用更灵活的SQL语句的结构来过滤。这可以让你使用规律的Python操作符来操作你映射的类参数：

```
>>> for name, in session.query(User.name).filter(User.fullname=='Ed Jones'):
...     print(name)
ed
```

`Query` 对象是完全可繁殖的（fully generative），意味着大多数方法的调用都返回一个新的 `Query` 对象（ a new Query object upon which further criteria may be added.）此对象仍可进行查询操作（不知道理解对不对），例如，你可以调两次 `filter()` 函数来查用户名为ed并且全名为Ed Jones的用户，相当于SQL中的AND操作：

```
>>> for user in session.query(User).\
...           filter(User.name=='ed').\
...           filter(User.fullname=='Ed Jones'):
...     print(user)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
```

常用过滤操作（Common Filter Operators）

这里是一份常用过滤操作的摘要：

- `equals`

```
query.filter(User.name == 'ed')
```

- `not equals`

```
query.filter(User.name != 'ed')
```

- `LIKE`

```
query.filter(User.name.like('%ed%'))
```

- `IN`

```
query.filter(User.name.in_(['ed', 'wendy', 'jack']))

# works with query objects too:
query.filter(User.name.in_(
        session.query(User.name).filter(User.name.like('%ed%'))
))
```

- `NOT IN`

```
query.filter(~User.name.in_(['ed', 'wendy', 'jack']))
```

- `IS NULL`

```
query.filter(User.name == None)

# alternatively, if pep8/linters are a concern
query.filter(User.name.is_(None))
```

- `IS NOT NULL`

```
query.filter(User.name != None)

# alternatively, if pep8/linters are a concern
query.filter(User.name.isnot(None))
```

- `AND`

```
# use and_()
from sqlalchemy import and_
query.filter(and_(User.name == 'ed', User.fullname == 'Ed Jones'))

# or send multiple expressions to .filter()
query.filter(User.name == 'ed', User.fullname == 'Ed Jones')

# or chain multiple filter()/filter_by() calls
query.filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')
```

注意 是 `and_()` 不是Python里的 `and` 操作符

- `OR`

```
from sqlalchemy import or_
query.filter(or_(User.name == 'ed', User.name == 'wendy'))
```

注意 是 `or_()` 不是Python里的 `or` 操作符

- `MATCH`

```
query.filter(User.name.match('wendy'))
```

注意 `match()` 使用 `MATCH` 或者 `CONTAINS` 来实现的，所以和数据库底层有关，在一些数据库下不能使用，比如说SQLite

查询返回的列表以及标量（Returning Lists and Scalars）

```
* ```all()```返回一个列表：
```

> > > query = session.query(User).filter(User.name.like('%ed')).order_by(User.id) query.all() [,] ```

- `first()` 对查询结果进行了一个限制-返回列表的第一个值：

```
>>> query.first()
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
```

- `one()` 完全匹配所以行，如果匹配不到，则返回一个错误，或者匹配到多个值也会返回错误：

```
# 多个值
>>> user = query.one()
Traceback (most recent call last):
...
MultipleResultsFound: Multiple rows were found for one()

# 匹配不到
>>> user = query.filter(User.id == 99).one()
Traceback (most recent call last):
...
NoResultFound: No row was found for one()
```

`one()` 对于那些希望分别处理查询不到与查询到多个值的系统是十分好的，比方说在RESTful API中，查询不到可能会返回404页面，多个结果则可能希望返回一个应用错误。

- `one_or_none()` 和 `one()` 很像，除了在查询不到的时候。查询不到的时候 `one_or_none()` 会直接返回None，但是在找到多个值的时候和 `one()` 一样。
- `scalar()` 援引自 `one()` 函数，查询成功之后会返回这一行的第一列参数，如下：

```
>>> query = session.query(User.id).filter(User.name == 'ed').\
...     order_by(User.id)
>>> query.scalar()
1
```

使用SQL语句查询（Using Textual SQL）

> > > from sqlalchemy import text SQL>>> for user in session.query(User).filter(text("id<224")).order_by(text("id")).all(): ... print(user.name) ed wendy mary fred ```

使用基于字符串的SQL语句（string-based SQL）可以通过冒号来指定参数。为参数复制可以使用 `params()` 来实现：

```
>>> session.query(User).filter(text("id<:value and name=:name")).params(value=224, nam
e='fred').order_by(User.id).one()
<User(name='fred', fullname='Fred Flinstone', password='blah')>
```

为了使用完全基于字符串的语句，可以使用 `from_statement()` 来实现。不需要额外的指定，完全的字符串SQL语句是根据模型（model）的名字来匹配的，如下，我们仅使用了一个星号就获取到了所有的信息（查找名字为ed的行）：

```
>>> session.query(User).from_statement(
...                     text("SELECT * FROM users where name=:name")).\
...                     params(name='ed').all()
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

在简单的例子中去根据名字匹配是可行的，但是当处理包含着重复的名字的复杂的语句的时候或者使用匿名的ORM架构的时候就不太现实了（unwieldy when dealing with complex statements that contain duplicate column names or when using anonymized ORM constructs that don’t easily match to specific names）。除此之外，有一个典型的表现就是在

我们匹配结果的时候，我们可能会发现处理找到的结果也是十分有必要的。在这种情况下，`text()` 架构允许我们把纯SQL语句与ORM映射对应起来，我们通过 `TextClause.columns()` 方法可以传一些表达式参数：

```
>>> stmt = text("SELECT name, id, fullname, password "
...                 "FROM users where name=:name")
>>> stmt = stmt.columns(User.name, User.id, User.fullname, User.password)
SQL>>> session.query(User).from_statement(stmt).params(name='ed').all()
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

当在 `text()` 架构里查找的时候，`Query` 可能会指定一些返回的实体，比如不返回全部的，只返回一部分信息：

```
>>> stmt = text("SELECT name, id FROM users where name=:name")
>>> stmt = stmt.columns(User.name, User.id)
SQL>>> session.query(User.id, User.name).\
...             from_statement(stmt).params(name='ed').all()
[(1, u'ed')]
```

计数（Counting）

```

```

> > > session.query(User).filter(User.name.like('%ed')).count() 2 ```

`count()` 函数是用来计算返回值里包含多少项的，SQLAlchemy总是把查询结果放到一个子集里边，`count()` 函数就是用来计算子集的。有些例子下面可以直接使用 `SELECT count(*) FROM table` 来计算有多少个值，但是SQLAlchemy并不去判断这是否是恰当的，毕竟准确的结果可以使用那个准确的方法去获得（however modern versions of SQLAlchemy don’t try to guess when this is appropriate, as the exact SQL can be emitted using more explicit means.）

需要我们对每一项分别计数的时候，我们需要在 `func` 模块里通过 `func.count()` 来直接指定 `count()` 函数，下面我们可以单独计算一下每个名称的数量：

```
>>> from sqlalchemy import func
SQL>>> session.query(func.count(User.name), User.name).group_by(User.name).all()
[(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]
```

为了实现 `SELECT count(*) FROM table` 的功能，我们可以这样做：

```
>>> session.query(func.count('*')).select_from(User).scalar()
4
```

如果我们直接根据 `User` 的主键来计算，那么 `select_from()` 可以去掉：

```
>>> session.query(func.count(User.id)).scalar()
4
```

Building a Relationship¶